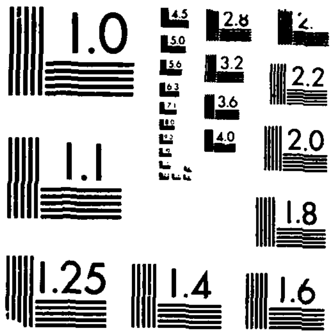
AD-A189 942   AN ANALYSIS OF MULTIPLE GRAIN CHIP ARCHITECTURES(U)   1/1
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
L SNYDER 28 FEB 87 N00014-85-K-0328

UNCLASSIFIED   F/G 12/6   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Final Report N00014-85-K-0328		5. TYPE OF REPORT & PERIOD COVERED Final 4/15/85 - 9/30/86
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE February 28, 1987
		13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Contd on pg 1

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel computation, multigauge architecture, parallellism, Quarter Horse multiprocessor, SIMD, MIMD, Poker Programming Enviroment, CHiP architectures, retargeting, Hearts, Systolic Array

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The major accomplishments of the Blue CHiP Project Multigrain Architecture Research are described in the areas of theory, algorithms, architecture software and VLSI.

Over Pg.

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

88 1 12 202

Final Report for Contract N00014-85-K-0328
An Analysis of Multiple Grain CHiP Architectures

Lawrence Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

1 Scope of Report

This contract, originally planned to cover three years, was concluded effective September 1, 1986 with the majority of the scientific questions resolved; only one year of funding was actually appropriated. This report recaps the scientific accomplishments during the actual period of the contract.

2 Multigauge Architecture

The concepts and issues covered by this contract were in the beginning poorly understood and quite amorphous. This makes for very interesting and exciting scientific work, but it also implies that there may be false starts. The first and probably the only noticeable one for us was in naming the concept: "Multigrain", though accurate and perspicuous for some scientists, was vague and ambiguous to others. The issue is that the concept of "granularity" is used many ways in parallel computation, and the term "multigrain" confused the matter further by adding one more. Accordingly, we created a new term "multigauge", which not only disambiguated it for existing concepts, it built on a useful analogy with railroads. "Multigauge" has been well received and seems to be being adopted. In any event we have used it throughout our work since early 1985, and will do so for this report.

2.1 Review of the Concept

A multigauge architecture is a design for a standard von Neumann machine that permits the data path to be partitioned into subunits that can execute concurrently when the data values are "small". Thus it is a method of achieving parallelism within the arithmetic/logic unit of a computer that is applicable to both stand alone serial processors and to the processor elements of highly parallel computers. Its advantages are the twin benefits of parallelism for narrow data and the ability to use the processor in its "regular mode" under programmer control when full precision is necessary. There is the added benefit that multigauging is available with minimal added hardware in certain cases, (explained below) but this is simply a benefit, not the justification for the architecture.

There is certain terminology and notation surrounding the multigauge concept that will be useful in the subsequent discussion: A standard von Neumann machine and a multigauge machine executing at full precision are said to be *wide gauge* machines with a datapath of B bits. A multigauge machine is partitioned into k *narrow gauge* machines, each executing a datapath of b bits; $B \geq k \times b$, and equality is generally assumed. Not all possible widths are necessarily available with every multigauge machine, so a multigauge machine is often described by listing the gauges with which it can execute; for example a (32, 16, 8) multigauge machine can be partitioned into two or four concurrent processors.

A multigauge machine can execute in either SIMD or MIMD modes. In the former, there is a single instruction stream and each narrow gauge machine has its own data stream. The MIMD case provides for an instruction stream and a data stream for each narrow gauge machine. Because it would be ludicrous to have an MIMD bit-serial machine, we postulate the *MIMD threshold*, the minimum value of b at which it is "reasonable" to run a multigauge machine in MIMD mode; we take $b=8$, though it may be larger.

These concepts and related work have been described in the literature [1]; a notable omission in the cited references is the work on TRAC, Texas Reconfigurable Array Processor [4]; this machine was a parallel processor capable of a certain level of multigauge processing.

2.2 Multigauging the Quarter Horse

To make our study of multigauging concrete, we applied the concept to a specific microprocessor architecture, the Quarter Horse [5]. This single chip, 32-bit microprocessor fabricated in 3μ CMOS technology was designed (in 90 days) at the University of Washington. It was thus a natural choice for multigauging experiments, because we understood it at the lowest levels of detail. Moreover, single chip microprocessors present serious problems for multigauging because of the limited "pin out", so the challenge was even greater. Before describing the impact of multigauging on the Quarter Horse, we give a very brief description of the architecture.

The Quarter Horse [5] has a 32-bit wide, dual-bus datapath, 32 general purpose registers, a Mead-Conway [6] ALU with Manchester carry-chain, a barrel shifter and PLA controller. The typical instruction requires 6 "microinstructions" and the design specification called for a 75ns clock cycle. The machine uses a self incrementing program Counter (PC). It also uses 32 pins each for connecting the memory address register (MAR) and the memory data register (MDR) to the memory system.

2.2.1 Partitioning the Components

Perhaps the first transformation necessary for multigauging a microprocessor that occurs to one, but probably the least complicated, is the partitioning of the datapath components to execute independently. In this section we review the changes required for the Quarter Horse, and comment on the difficulty of partitioning elements not found in the Quarter Horse.

The register file is trivially partitioned for normal multigauge execution, since each bit is independent. Nothing has to be done. If, however, more exotic instructions are provided, then some additional circuitry may be necessary [3]. Neither the MAR nor the MDR require any change.

The Mead-Conway ALU requires modest modification. For all but one of the narrow gauge datapaths, carry-in logic must be added. Symmetrically, all but one narrow gauge machine needs flag logic, if it is to be executed in MIMD mode. (For SIMD execution we assume only one of the narrow gauge processors can influence the control (branch), and so only one machine needs flag logic; this might as well be the wide gauge machine's flag logic.) Finally, the carry chain must be segmented between each narrow gauge machine, but because the design of the Manchester carry chain provides drivers every four bits, this change is very straightforward.

The PC need not be split for SIMD execution and for MIMD execution, the problem is analogous to splitting the ALU.

To partition the barrel shifter requires a rather surprising modification based on the following fact: A barrel shifter of b-bits in width requires a height of $O(b)$ bits. Thus, partitioning the shifter in half requires a shifter only half the height, and so each of the two narrow gauge shifters requires only a quarter of the area of wide gauge shifter. Other narrow gauges take correspondingly less area. This phenomenon, which is based on the $O(n^2)$ growth rate of the area of barrel shifters, and which is predicted by the theory [6], implies that both vertical and horizontal wires must be segmented at each "narrow gauge boundary" in each dimension. However, the definition of boundary is different in each dimension [7].

Other typical data path components, though not found in the Quarter Horse, tend to fall into one of the three classes already identified: "bitwise independent" components requiring no change, e.g. complementor, "linear" components requiring straightforward segmentation, e.g. a counter, and "quadradic" components requiring two dimensional segmentation, e.g. a multiplier.

2.2.2 Memory System Partitioning

When a wide gauge machine is partitioned into narrow gauge machines, the size of the data address can be dramatically reduced. If this means that the size of the addressable memory is also reduced, then multigauging will not be a desirable way to exploit parallelism. (Recall that dividing the address in half reduces the *address space* by dividing its *log* in half; e.g. a 32 bit address refers to over four billion locations, but a 16-bit address has roughly sixty-five thousand.) Although it would not quite solve this addressing problem, adding more wires between the processor and the memory would seem to offer the right "fix." But it is definitely prohibited by the fact that single chip processors will not have enough "pins" from the "package".[1] Obviously, the von Neumann bottleneck gets a lot narrower in the presence of multigauging.

Fortunately, there is a rather straightforward solution, that sacrifices very little. The technique is to provide segment registers on the "memory side" of the memory-processor interface for each narrow gauge machine. Each register contains the high order bits of the narrow gauge machine's address, and for the "larger" gauges, e.g. 8 bits or greater, provides adequate addressing capability in each segment. Special instructions are provided to load the segment registers, which may require more than one narrow gauge operation. If the machine is to be executed in MIMD mode, it is advisable to have both an instruction stream and a data stream segment register for each multigauge device.

As a benefit for supporting addressing when the segments are "smallish", it is advisable to provide an "add with carry to memory" instruction for computing a sequential address. The idea is that when the segment addresses are computed, there should be a smooth mechanism for crossing the segment boundary, or at least recognizing when it has been crossed. This will add flag logic to the SIMD case, but it is our estimate that it is more effective than depending exclusively on the software to manage segment boundaries.

2.2.3 Control

Perhaps the most difficult problem in multigauging is the control of the various machine forms [2]. To simplify the matter, we consider SIMD and MIMD separately.

The ideal situation, available only to the SIMD case basically, is to use a single controller for the wide gauge and all of the narrow gauge machines. This requires that each machine have essentially the same instruction set. The obvious exceptions are the gauge shifting instructions which only apply in one gauge, and control instructions which have somewhat different semantics in different gauges. A single controller is feasible because the control lines generally run "perpendicular" to the datapath.

MIMD multigauge machines have multiple program counters and thus execute different instructions at the same time. The control implication of this fact is that different control lines must be set for each narrow gauge machine. This probably means that there are separate controllers for each narrow gauge machine. It would be possible in the microprogrammed case to use a multiported control ROM with separate microPCs for each narrow gauge machine; the problem is that a careful design would be needed because microcode sequencing is much more complex than normal program sequencing.

The final observation about MIMD control concerns the problem that the instruction streams and the data streams should have their own segment registers for each narrow gauge machine, and that in MIMD execution the narrow gauge machines can be executing different instructions at any given moment. This means that one machine might be fetching an instruction and another might be fetching data; the memory system must be told which

[1] Exotic packaging helps but doesn't solve the problem, so we face the problem headon.

segment register to use in each case. This can be done using additional control signals between the processor and memory, but if the prohibition against additional wires between the processor and memory is in force, then we will be required to observe a protocol in memory reference. The rigid reference ordering can be implemented independently on each side of the von Neumann bottleneck and thus no additional wires are needed. Use of the protocol is no real problem, because many RISC machines - the kind for which the prohibition can be expected to apply - use a regimen in which memory references follow a rigid protocol.

2.3 Applications

The possibility of creating a multigauge machine can be seen rather clearly from the foregoing remarks. What is not entirely evident is whether there is any beneficial speedup from multigauging. The matter is complicated by the fact that certain multigauge machines can be implemented with little or no additional hardware, but these are the rigid, SIMD machines with only one or a few narrow gauge sizes. It is not clear how useful such a machine might be. On the other hand, a full, MIMD multigauge machine with many gauge sizes would be much more useful and require a lot more hardware. This hardware may be used better for other purposes. Finally, multigauge machines are subject to Amdahl's law, i.e. a substantial amount of any given computation may require the wide gauge and thus yield no speedup. In order to determine which of these possibilities was truth and which was conjecture, we analyzed some specific applications[3]. The results were impressive.

Two applications have been studied in depth in order to ascertain the amount of speedup that can actually be achieved. The applications both come from graphics: Bézier curve generations and scan conversion using Bresenham's algorithm. Since no multigauge machine has actually been built, it has been impossible to obtain true measurements. Nevertheless, our methodology is sufficiently detailed that we can confidently report the findings as the next best thing to actual measurements.

The methodology is to seek problems of practical interest that compute with "narrow" data; for graphics we consulted Professor Tony DeRose. Next the problems were programmed in C and tested on realistic data (on a VAX) to produce actual dynamic behavior. Then, the C source programs were compiled for the Quarter Horse microprocessor whose object code can be put into one-to-one correspondence with the test program code. The difference is, of course, that the "narrow" gauge instructions can be executed several at a time. Then the machine instructions of the Quarter Horse object program were expanded to their microcode equivalents. Each of these microinstructions has a fixed duration determined by the clock rate of the computer. Consequently, we can describe the behavior of the machine in a way that can be reported either as a unit free speedup (the ratio of the base machine's performance over the multigauge machine's performance) or in absolute seconds.

The results are striking [3]. For a Bézier curve $Q(u)$ of degree n defined by

$$Q(u) = \textstyle\sum_{i=0}^{n} V_i B_i^n(u), \quad u \epsilon\ [0, 1], \qquad (1)$$

where $V_0,\ldots, V_n$ are controlling points commonly called control vertices, and $B_0^n, \ldots, B_n^n(u)$ are the n^{th} degree Bézier blending functions defined by

$$B_i^n(u) = \binom{n}{i} u^i(1-u)^{n-i},$$

we can perform the computation for a 1024 $\times$ 1024 pixel display (i.e. 10 bits precision on output) with $b = 16$ bits of precision internally, which is required to control error propagation. Thus a $k = 2$ multigauge machine can achieve speedups η for n points in the range $[64, \infty]$

$$\eta \in [1.937, 1.939].$$

Of course, $k = 2$ is the theoretical best. It is important to observe that the Bézier curve evaluation must change gauge for each generated point, an overhead many algorithms wouldn't have, and still it approaches optimal speedup.

To get an idea of how multigauging would apply to a broad class of graphics problems, we studied line segment transformations and scan conversion. This is a "typical" graphics problem in that it uses a 4 $\times$ 4 homogeneous matrix to transform points, a very common 'subroutine'.

Bresenham's algorithm was performed by a $k = 2$ multigauge machine that generates points for a line "from both ends towards the middle", a scheme that lead to a novel architectual feature, the virtual register[3]. Using the kind of analysis mentioned above, we find

$\eta = 1.70$	for a single line,
$\eta = 1.99$	for 50 lines, and
$\eta = 1.9995$	for 1000 lines.

The latter case, of course, is most typical for graphics displays of "interesting" objects.

3

A variety of other topics were treated in addition to the multigauge work, as outlined in the original proposal. In this section we detail the results of the most significant ones.

3.1 Poker

The Poker Parallel Programming Environment is the first complete set of programming support facilities developed expressly for parallel computation. Developed by the Blue CHiP project in research conducted continuously since January 1982, Poker provides a novel setting for programming nonshared memory parallel computers. Because the concepts are completely new and differ substantially from all previous programming languages, the amount of research that has been expended on Poker has been large. Nevertheless, concepts in Poker apply directly to the application of multigauging in parallel architectures, and thus Poker research was an important adjunct to the multigauge research. We review the specific studies supported by this contract.

3.1.1 Initial Distribution of Poker

As just indicated, Poker contains many novel ideas that were motivated by the realization that parallel programming is substantially harder than serial programming and that programmers will require greater support from their programming environment if they are to use parallel computers productively. Like any piece of research some of the ideas in Poker are fundamental and important, some are misguided and wrong-headed and most are in between. These are the ones that need scrutiny by the research community, but the only way for them to get a clear picture of the system is to use it. Accordingly we spent considerable time preparing Poker for distribution, and in October of 1985, Poker was released to the research community[8]. There has been a steady flow of requests for the system ever since.

3.1.2 Retargeting Poker

Poker was originally developed to program the CHiP architectures generally and the Pringle Parallel Computer (a hardware simulator of CHiP architectures), specifically. As a result many of its features were specialized to the CHiP architecture. When it became clear that Poker could be a far more general facility than just a programming language for the CHiP Computer, that it could be used for any of the known parallel architectures, then it became critical to study two problems: First, how could the features specialized to the CHiP architecture be removed and replaced with features with wider applicability, and second, how could Poker be restructured to generate programs for arbitrary parallel machines, i.e. how could Poker be retargeted? Solutions to both problems have been developed under this contract.

Recognizing Poker features that are peculiar to the CHiP architecture was easier than replacing them with alternatives. Examples of CHiP-specific features are (refer to the Poker description [8]): The switches (circles) in the switch setting view, the use of the XX programming language for specifying processor element codes, the geometric rather than the topological layout of the interconnection graph, the convention that the number of processors be a power of 4, etc. The switches were removed simply by not drawing them; they still play a valuable role that would require a similar technique if they were not available, so it was deemed reasonable to just keep them in place but hidden. The XX programming language remains in the system, but the ability to use C as a processor element programming language was added. Nothing was done about the geometric emphasis of the communication graph; it was just too fundamental a part of the system and is not too great of an impediment to others. Clearly, any subsequent reimplementation would remove this geometric emphasis in favor of a more topological style. The "power of four" problem is simply ignored since this is not a serious restriction (most parallel computers have a similar character) and the logical processors used by Poker can always be ignored. Other CHiP-specifics have been handled using similar "mixed" strategies.

The real challenge to the Poker research team was the retargeting of Poker to other parallel architectures[9]. Retargeting, or porting, sequential software is still a difficult task a quarter of a century after the problem was first appreciated; it is the primary motivation why computer manufacturers retain extinct instruction sets for their computers – they need the existing software base to continue to run. The problem is IMMENSELY more difficult in parallel computation because the architectures "show through" into the programs far more in parallel computation than in serial computation. Of course, there is no base of parallel software, but if it is to be developed, the retargeting problem must be solved.

A major accomplishment of this contract was the retargeting of Poker to a new machine[9]. To be specific, the restructuring of Poker in order that it be able to compile code for different machines was accomplished by the current contract and the actual porting of Poker to the CalTech Cosmic Cube was begun as a proof of the concept. (The port was completed under another contract.) The three major requirements to enable Poker to generate code for another parallel computer are (1) the development of a C compiler for the processor element codes, (2) the development of a runtime environment for the Poker programs on the host machine, and (3) the construction of a "mapper" that converts the communication structure used in Poker into an appropriate communication structure for the host machine. The first requirement is almost always met by the computer manufacturer by industry tradition, and the second can likely build effectively on the available software provided by the manufacturer. For the Cosmic Cube, these three constituents required approximately 3 man months to get up and running.

3.1.3 Hearts: Poker for Systolic Arrays

The Poker Parallel Programming Environment is a general facility for programming arbitrary MIMD parallel computers. It is common to use systolic arrays as subroutines when engaged in such programming, but doing so is somewhat cumbersome because of the generality of Poker. The question arises whether Poker could be made more expressive if it was limited to programming only systolic arrays. This is a reasonable restriction since there are many systolic array users and so such a "restricted" system could still find wide applicability. We have considered the question [10,11] and found that such a system would not only be feasible, but it would likely be very convenient.

3.2 Parallel Programming Paradigms

There are many programming paradigms known for serial programming – divide-and-conquer, greedy, dynamic programming, etc., but because parallel programming is so new, there are few "standard" problem solving methods known. A graduate student, Phil Nelson, has studied this topic for his doctoral research in an effort to identify a set of parallel programming paradigms. He has produced a list including systolic and pipelined algorithms, divide-and-conquer, and a new class called the CAB class, for compute, aggregate and broadcast.

One reason to study paradigms is as an aid to discovering new algorithms: A technique that worked well once might be applied in another context with good results. Using this approach, Nelson has discovered a new matrix multiplication algorithm that is based on the divide-and-conquer paradigm[12]. This new algorithm uses the same principles of Strassen's optimal serial matrix multiplication algorithm in that it subdivides the matrix product problem until it reduces to a large set of 2×2 matrix products. It should be noted that Nelson's algorithm doesn't eliminate any multiplication operations and so is not subject to the instablity problems that Strassen's algorithms is. (This result was reported at the SIAM meeting in Norfolk, VA.)

Another reason to study paradigms is that transformations that one wishes to apply to an algorithm might well be applicable to all of the algorithms in its paradigmatic class. Contraction is one such operation which has been studied in this way[13]. Recall that contraction is required when a parallel program has many more parallel processes than the parallel computer has parallel processors, and so multiple processes must be allocated to each processor. It has been shown that algorithms of the CAB paradigm can be contracted using the same allocation scheme.[13]

4 Conclusions

It is evident from the research summarized in this report that multigauging is an effective scheme for realizing parallel speedup of processors when the data being processed is "small". The best results come for the many problems that admit an SIMD execution policy, since the SIMD multigauging capability can be added to a standard computer or parallel computer processing element with little hardware cost. In this case multigauging provides essentially free speedup. The MIMD case is considerably more expensive in terms of hardware and its potential applications turned out to be unexpectedly rare. It can therefore be recommended that an SIMD multigauge machine be constructed along the lines discussed in the reports [2,3].

5 References

(* Means supported by the contract.)

*1. An Inquiry into the Benefits of Multigauge Computation, L. Snyder, *Proceedings of the International Conference on Parallel Processing*, IEEE, 1985, pp. 488-492.

2. An Investigation into the Design Costs of a Single Chip Multigauge Machine, L. Snyder and C. Yang, *Technical Report*, 86-06-01, University of Washington, 1986.

3. Near-Optimal Speedup of Graphics Algorithms Using Multigauge Parallel Computers, T. DeRose, L. Snyder and C. Yang, *Technical Report*, University of Washington, February, 1987.

4. An Overview of the Texas Reconfigurable Array Computer, M.C. Sejnowski, E. T. Upchurch, R.N. Kapur, D.P.S. Charlu and G. J. Lipovski, *AFIPS Proceedings NCC*, 1980, pp. 631-641.

5. The Quarter Horse I: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip, S. Ho, B. Jinks, T. Knight, J. Schaad, L. Snyder, A. Tyagi and C. Yang, *Proceedings International Conference on Computer Design: VLSI in Computers*, IEEE, 1985, pp. 261-266.

6. J. Vuillemin, Combinatorial Limit to the Computing Power of VLSI Circuits, *IEEE Syposium on Foundations of Computer Science*, IEEE Computer Society, *1980.*

7. An Investigation of Multigauge Architectures, C. Yang, Ph.D. Thesis, University of Washington, 1987 (to appear).

*8. Poker (3.1) Reference Guide, *Technical Report*, 85-09-03, University of Washington, 1985.

*9. Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment, L. Snyder and D. Socha, *Proceedings of the International Conference on Parallel Processing*, pp. 628-635, 1986.

*10. Programming Environments for Systolic Arrays, *Proceedings of O-E LASE-86, Society of Photo-Optical and Instrumentation Engineers*, Vol. 614, Keith Bromley (ed.), 1986, pp. 134-144.

*11. Hearts: A Dialect of the Poker Programming Environment Specialized for Systolic Computation, *Systolic Arrays*, W. Moore, A. McCabe, and R. Urquhart (eds.) Adam Hilger, Bristol, pp. 71-80, 1987.

*12. A Non-systolic Matrix Product Algorithm, P. Nelson, *Technical Report*, University of Washington, 85-11-02, November, 1985.

*13. Programming Solutions to the Algorithm Contraction Problem, L. Snyder and P. Nelson, *Proceedings of the International Conference on Parallel Processing*, 1986, pp. 258-261.